FAUSTLIVE
Just-In-Time Faust Compiler... and much more

Sarah DENOUX and **Stephane LETZ** and **Yann ORLAREY** and **Dominique FOBER**
GRAME
11 Cours de Verdun (GENSOUL)
69002 Lyon
FRANCE
{sdenoux, letz, orlarey, fober}@grame.fr

Abstract

FaustLive is a standalone just-in-time FAUST compiler. It tries to bring together the convenience of a standalone interpreted language with the efficiency of a compiled language. Based on *libfaust*, a library that provides a full in-memory compilation chain, FaustLive doesn't require any external tool (compiler, linker, etc.) to translate FAUST source code into binary executable code.

Thanks to this technology, FaustLive provides several advanced features. For example it is possible, while a FAUST application is running, to modify its behavior on-the-fly without any sound interruption. It is also possible to migrate a running application from one machine to another, etc.

Keywords

Audio, FAUST, DSP programming, remote processing and interfacing

1 Introduction

FAUST [Functional Audio Stream] [6] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing music languages like Max, PD, Supercollider, etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to C/C++ to implement efficient sample-level DSP algorithms.

But, if compilers have the advantage of efficiency, they have their own drawbacks compared to interpreters. Compilers traditionally require a whole chain of tools to be installed (compiler, linker, development libraries, etc.). For non-programmers this task can be complex. The development cycle, from the edition of the source code to a running application, is much longer with a compiler than with an interpreter. This can be a problem in creative situations where quick experimentation is essential. Moreover, binary code is usually not compatible across platforms and operating systems.

FaustLive is an attempt to bring together the convenience of a standalone interpreted language with the efficiency of a compiled language. Based on *libfaust*, a library that provides a full in-memory compilation chain, FaustLive is a standalone application that doesn't require any external tool to translate FAUST source code into binary executable code and run it. In many aspects FaustLive behaves like a FAUST interpreter with a very short development cycle (not very different, in that aspect, from modern compiled LISP environments, or from the approach presented by Albert Graef with Pure in [1]).

Moreover, FaustLive provides some advanced features to speedup the development cycle. For example, while a FAUST application is running, it is possible to edit and recompile its FAUST code on-the-fly, without any sound interruption. If the application is using JACK as driver, all audio connections are maintained. Another interesting feature is the possibility to migrate a running application from one machine to another through the network even across operating systems. Applications can also be controlled remotely, using HTTP or OSC.

FaustLive offers a lot of flexibility to prototype audio applications. It can also be connected to FaustWeb, a remote compilation service to export the application as a traditional binary for one of the various operating system and audio architecture supported by the FAUST ecosystem.

Since FaustLive is based on *libfaust*, the FAUST compiler project will first be presented (see Section 2). Then FaustLive will be shortly described through a typical use case (see Section 3) to finally be detailed over its technical aspects (see Section 4).

2 Faust Compiler

The FAUST compiler translates a FAUST program into an equivalent imperative program (typically C, C++, Java, etc.), taking care of generating efficient code. The FAUST package also includes various architecture files, providing the glue between the generated code and the external world (audio drivers and user interfaces).

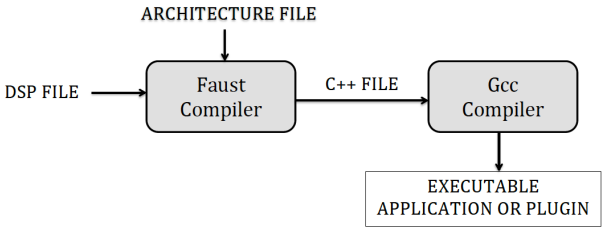

Figure 1: Steps of FAUST compilation chain

The current version of the FAUST compiler produces the resulting DSP code as a C++ class, to be inserted in the architecture file. The resulting C++ file is finally compiled with a regular C++ compiler to produce the final executable program or plug-in (Figure 1).

The resulting application is structured as shown in Figure 2. The DSP has become an audio computation module. As for the architecture, it turned into links to the user interface and the audio driver.

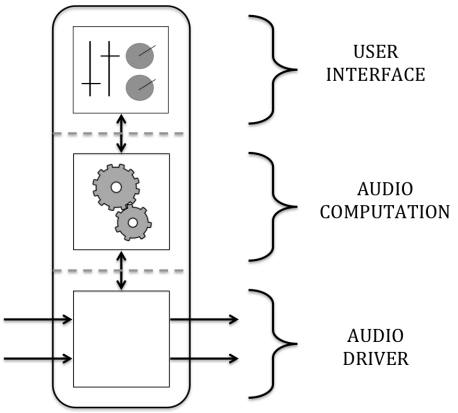

Figure 2: FAUST application structure

2.1 LLVM

LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure, designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages. Executable code is dynamically produced using a "Just In Time" compiler from a specific code representation, called LLVM IR[1]. Clang, the "LLVM native" C/C++/Objective-C compiler is a front-end for LLVM Compiler. It can, for instance, convert a C or C++ source file into LLVM IR code (Figure 3).

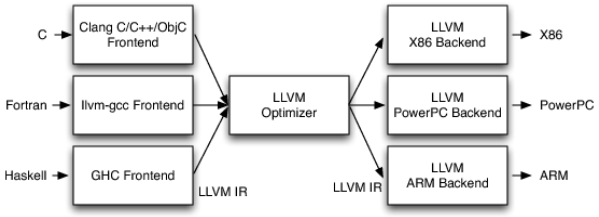

Figure 3: LLVM compiler structure

Domain-specific languages like FAUST can easily target the LLVM IR. This has been done by developing a special LLVM IR backend in the FAUST compiler, [5].

2.2 Dynamic compilation chain

The complete chain goes from the DSP source code, compiled in LLVM IR using the LLVM backend, to finally produce the executable code using the LLVM JIT. All steps are done in memory. Pointers on executable functions can be retrieved in the resulting LLVM module, and their code directly called with the appropriate parameters.

In the *faust2* development branch, the FAUST compiler has been packaged as an embeddable library called *libfaust*, published with an associated API, [2]. This API imitates the concept of oriented-object languages, like C++. The step of compilation, usually executed by gcc, is accessed through the function *createDSPFactory*. Given a FAUST source code (as a file or a string), the compilation chain (FAUST + LLVM JIT) generates the "prototype" of the class, called llvm-dsp-factory. Then, the function *createDSPInstance*, corresponding to the "new className" of C++, instantiates a llvm-dsp. It can then be used as any object, run and be controlled through its interface.

Embedding this technology in a program or a plug-in enables dynamic modifications of the audio computation module of a FAUST application [4].

[1] *The Intermediate Representation* is an intermediate SSA representation

3 FaustLive - Use Case

FaustLive is a QT-based[2] software that permits to launch FAUST applications from their source code without having to precompile them (Figure 4).

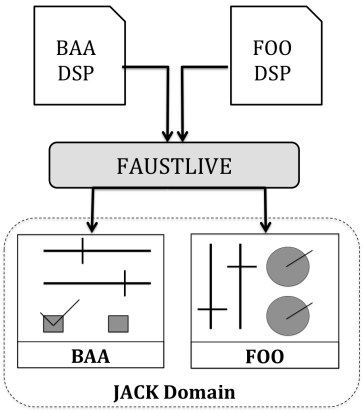

Figure 4: FaustLive principle

FaustLive exploit dynamic compilation, associated with multiple interfacing systems and audio drivers to modulate the structure of FAUST applications and simplify FAUST prototyping process.
To give an idea of FaustLive's potential, the following section presents its diversified features, showing the corresponding alterations in the structure of the applications.

The starting point of FaustLive's features is drag and drop. A FAUST DSP can be opened in a new window or it can be dropped on a running FAUST application. As a result, an intermediate state emerges in which the two applications coexist. The arriving application copies the established audio connections. Then, the output of the old application is cross-faded to the new one (Figure 5). At last, the dropped application durably replaces the previous one. With that system, a running application can be changed endlessly, without audio click.

This mechanism also allows source edition. When the user chooses to edit its FAUST code, it is opened in a text editor. And as his changes are saved, the application is updated using the crossfade mechanism (Figure 6).

[2] *QT* is a framework for interface design

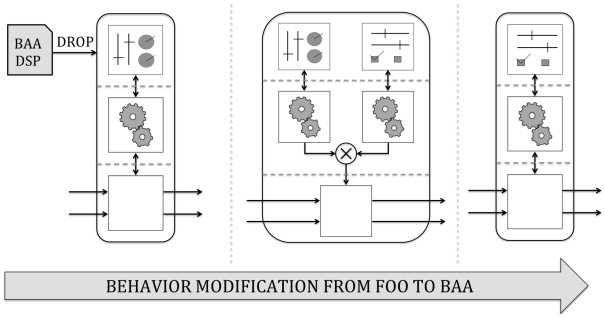

Figure 5: Behavior modification

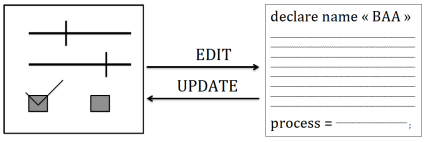

Figure 6: Dynamic source edition

JACK was primitively adopted as audio driver for it allows the user to connect its FAUST applications between themselves. Other drivers have then been added, making this component of the structure as flexible as the others. So when FAUST applications are running, FaustLive gives the possibility to dynamically switch the audio driver. FaustLive does not need to be stopped. The migration is made during execution and is applied to every FAUST application running. JACK, NetJack, CoreAudio and PortAudio are the integrated drivers in FaustLive (Figure 7).

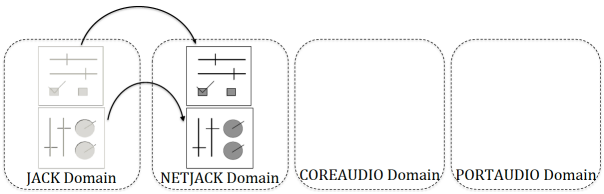

Figure 7: Dynamic driver migration

FaustLive expands its radius of action to external interactions. A smartphone can open an OSC[3] interface, controlling the application remotely (Figure 8).
Likewise, a HTML interface is accessible through a Qr Code[4]. By scanning it with a

[3] *OSC* : Open Sound Control
[4] *QR code* (abbreviated from Quick Response Code) is the trademark for a type of matrix barcode (or two-

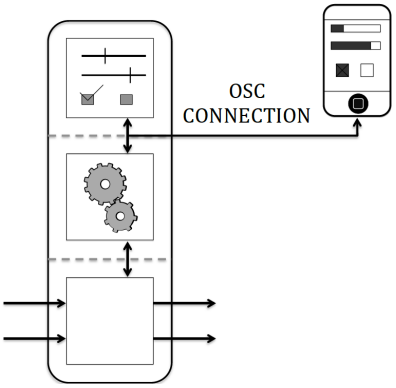

Figure 8: OSC interface

touchpad (for instance), the remote interface is opened in a browser. In both cases, the interface is duplicated and a synchronization between the local and remote interface is established.

The HTML interface has an additional interest: it is set up to enable drag and drop. Therefore, the user controlling the remote interface can also change the behavior of the application by dropping his own DSP. It is sent to the local application where it replaces the running one, using the crossfade mechanism. Finally, the remote interface is updated (Figure 9).

If many or/and heavy FAUST applications are opened, local CPU load can be saturated. The migration of calculations to other machines can lighten this load. On account of dynamic compilation, the audio computation module can be relocated on another machine (Figure 10). The list of remote servers available is built dynamically so that it is simple to switch from local processing to remote processing.

A user may wish to run his FAUST application in an other environment (Max/MSP, SuperCollider, ...). For that matter, a link to FaustWeb, a remote compilation web service, is integrated in FaustLive. The user only has to choose the platform and environment he wishes to target. In return, he will receive the binary of the requested application or plugin.

When FaustLive is exited, the last configuration is saved and will be restored at its next

dimensional barcode)

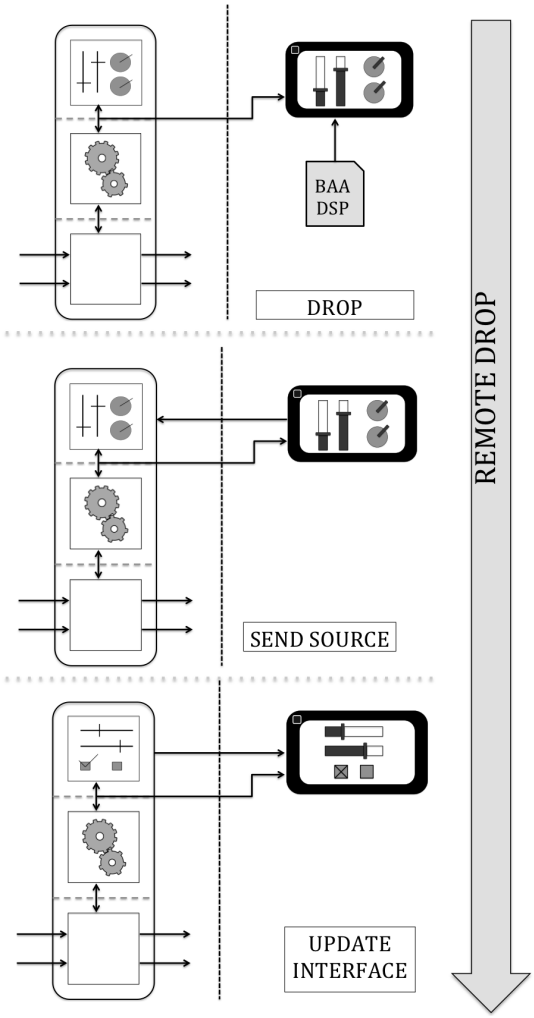

Figure 9: Remote drop

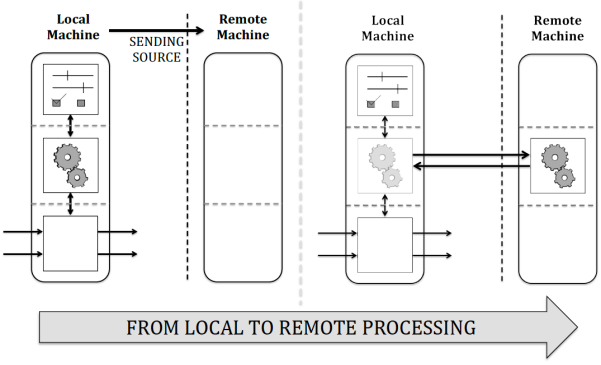

Figure 10: Remote processing

execution. A user may also save the state of the application at any moment. In a second phase, he will be able to reload his snapshot, by importing it in the current state or recalling it (Figure 11).

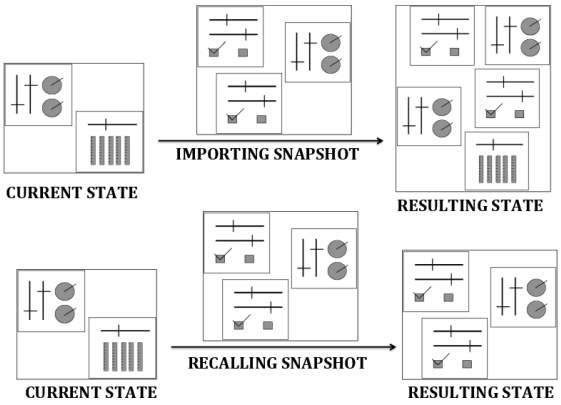

Figure 11: Reloading snapshot

4 FaustLive - Technical View

4.1 Basic FaustLive Features

The first aim of FaustLive is to create a dynamic environment for FAUST prototyping, by embedding *libfaust*. The resulting dynamic compilation chain (Figure 12) presents the advantage of speeding up the compilation process. Returning almost right away the executed application, this compiler is a stepping stone for dynamic behaviors.

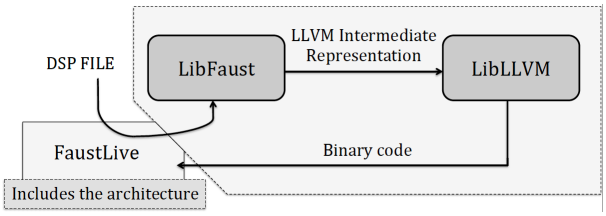

Figure 12: Compilation chain in FaustLive

Now that it is possible to dynamically compile FAUST code, new prospects are rising. A user may drop his FAUST code as a file, a string or a url, on a running application. As a result, the code is immediately given to the embedded compiler and the new application replaces the previous one. Since FaustLive is designed for dynamic uses, it is very important to ensure a continuity in the sound. For that matter, a crossfade is calculated between the two relaying FAUST applications.

Moreover, a FAUST application is linked to its source, so that any modification in the FAUST code will lead to a recompilation. This particular aspect is central, for it simplifies the prototyping process: a user can modify his code at leisure and see/hear instantly the result.

An important asset of FaustLive is the coexistence of multiple FAUST applications, in opposition with the QT-JACK architecture from FAUST "static" distribution, where every FAUST program has to be compiled separately to produce its own application. Here, each application evolves with the actions it undergoes and has its own set of dynamic parameters (Figure 13).

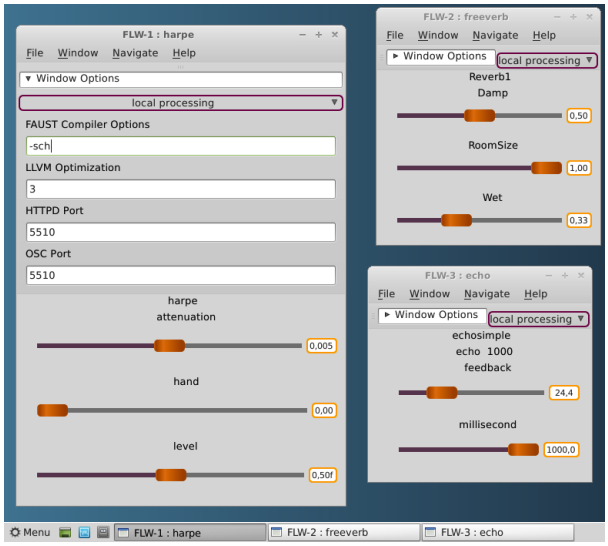

Figure 13: FaustLive's environment

4.2 Audio Drivers

FaustLive has integrated JACK, CoreAudio, NetJack and PortAudio[5]. So that it's possible to switch audio structures or modify its parameters (such as buffer size or sample rate) during FaustLive's execution. Every running audio client is stopped, then the applications are transferred in the new domain to finally be restarted.

4.2.1 JACK

JACK is a system for handling real-time, low latency audio (and MIDI). It runs on GNU/Linux, Solaris, FreeBSD, OS X and Windows. It can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves.

[5]JACK, CoreAudio and NetJack are used on OSX, JACK and NetJack on Linux, PortAudio, JACK and NetJack on Windows.

Therefore, an interesting constraint in using JACK is the matter of the connections. When connections have been established, the objective is to maintain them even if the FAUST application changes in a window. If the new application has more ports than the previous one, the user will have to make the connections himself.

4.2.2 NetJack

NetJack is a Realtime Audio Transport over a generic IP Network, fully integrated into JACK. NetJack synchronizes all clients to one soundcard, so there is no resampling or glitches in the whole network. The master imposes the sample rate and buffer size, in relation to its audio device.

4.2.3 CoreAudio and PortAudio

Because the protocol has to be strictly the same on the client and on the server's side, JACK and NetJack have to be linked as a dynamic library. The problem it brings is that FaustLive's installation is linked to JACK's installation. To avoid this inconvenience for beginner users, a CoreAudio[6] and PortAudio[7] versions have been developed. Included in the standard libraries or easily linked as a dll, they do not expand the user's work.

4.3 Control Interfaces

To offer a modular application, FaustLive expands the choices of the user, concerning the control interface.

4.3.1 OSC Interface

OSC protocol is integrated to FaustLive to offer another type of interface and enable interoperability. Many audio environments and devices implement this protocol so that FaustLive will be able to communicate with them. The user can configure the port on which the protocol is started and then control the interface with, for instance, an OSC touch application.

[6] *CoreAudio* is the digital audio infrastructure of iOS and OS X. It provides a framework designed to handle audio needs in applications.

[7] *PortAudio* is a free, cross-platform, open-source, C/C++ audio I/O library. It is intended to promote the exchange of audio software between developers on different platforms.

4.3.2 HTML Interface

FAUST HTML interface is also a component of FaustLive. Loaded on any browser, this interface controls the DSP's parameters, through a HTTP connection. When it is built, a server is started, taking care of delivering the HTML page (Figure 14). A synchronization between the local and the remote interface is also insured.

To ease the opening of the interface, a Qr Code is built from the HTTP address, thanks to *libqrencode*. Most smartphones and portable equipments have a QrCode decoder. By scanning the Qr Code, a browser gets connected to the interface page.

4.3.3 Preferences

The challenge FaustLive was confronted with is to provide an interface that gives as many liberties as possible to the user all the while being easy to apprehend. In that direction, OSC and HTTP ports are configurable in the window's options. The window toolbar is collapsed, by default, so that a "basic" user won't feel assailed by preferences (Figure 13). Both protocols use 5510 as default port. When the TCP listening port number is busy, the system automatically looks for the next available port number.

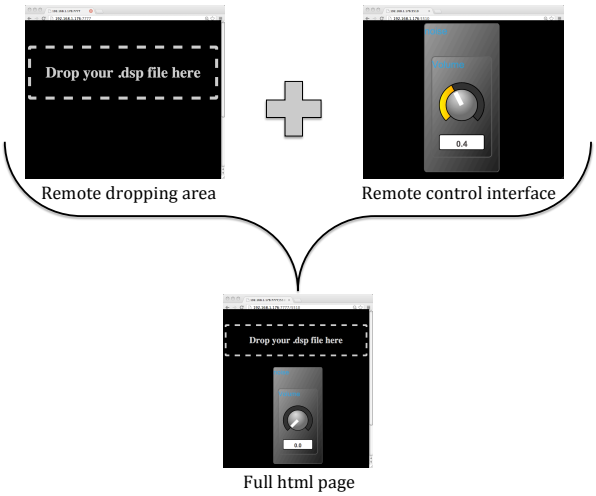

Figure 14: HTML interface with control and dropping services

4.3.4 Remote Drag and Drop

As the rest of FAUST current distribution, the HTML interface has a "static" behavior.

The intention, to copy FaustLive's dynamic behavior, led to adding a dropping area to the HTML interface. This HTTP service is independent and specific to FaustLive. The server, started by FaustLive, is able to create a HTML page that encapsulates the remote interfaces. The resulting service of remote interface and DSP drop has the following address: http://IP:DroppingPort/InterfacePort (Figure 14).

The dropping port is set in the preferences and is common to all the FAUST applications. The remote interface port is distinct for every FAUST application and editable in the window's options.

The reaction to the drop follows FaustLive's model. The DSP is first sent to FaustLive as a HTTP post request. The DSP is compiled and replaces the previous one, after the crossfade. At last, the remote interface is updated.

4.4 Remote Processing

To widen its benefits, FaustLive enables remote processing. The compilation and process calculation are redirected on a remote machine and local CPU load can be lightened.

On a remote machine, an application starts a HTTP server, offering the remote compilation/processing service. This server is waiting for requests.

On the client's side (FaustLive), an API "proxy" makes it transparent to create a remote-dsp rather than a local llvm-dsp (c.f 2.2). This API, *libfaustremote*, takes care of establishing the connection with the server.

The first step (compilation) is carried out by the function *createRemoteDSPFactory*. The code is sent to the server, which compiles it and creates the "real" llvm-dsp-factory. The remote-dsp-factory returned to the user is an image of the "real" factory. Before sending the FAUST code, a FAUST to FAUST compilation step is executed locally, to solve all the dependencies. This way, the expanded code sent to the server is self-contained.

The remote-dsp-factory can then be instantiated to create remote-dsp instances, which may run in the audio/visual architecture chosen (here, FaustLive).

To be able to locally create the interface, the server returns a json-encoded interface. This way, the function *buildUserInterface* can be recreated, giving the impression that a remote-dsp works as a local llvm-dsp.

Moreover, the audio processing is redirected through a NetJack connection. The audio data is sent to the remote machine which processes it and sends back its results. In addition to the standard audio flow, one midi port is used to transfer the controllers values (Figure 15). The benefit of this solution is to transmit synchronized audio and controllers in the same connection. Moreover, the audio samples can be encoded using the different possible audio data types : float, integer, and compressed audio (using the OPUS codec[8]).

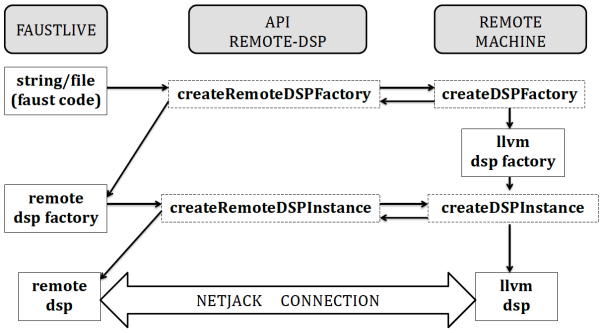

Figure 15: Remote compilation

libfaustremote uses *libcurl* to send http requests to the remote server, handled with *libmicrohttpd*.

On FaustLive's windows, the service of remote processing is simply interfaced. The Zero-Conf protocol is used to scan the remote machines presenting the service. A list is then dynamically built with the available ones. By browsing in the list, the user can then switch from a machine to another or come back to local processing very easily.

4.5 FaustWeb

In order to simplify the accessibility of the FAUST compilation, this web service of remote compilation has been conceived. It receives a FAUST DSP and returns a plugin or application in the chosen target architecture. As an outcome, the installation of FAUST package and all additional SDKs on the user machine is not necessary anymore. Anyone can write a FAUST

[8]http://www.opus-codec.org

application, send it to the server and receive a plugin.

This service is accessible from a browser but requires several requests. Through FaustLive, the export is facilitated. A menu is dynamically built with the platforms and architectures available. And as the user makes his choice, his code is sent to the server. The first step is the syntax verification, returning a sha1 key, with which multiple requests can be made. The second step is the compilation, using the standard "static" chain and returning the chosen application to the user (Figure 16).

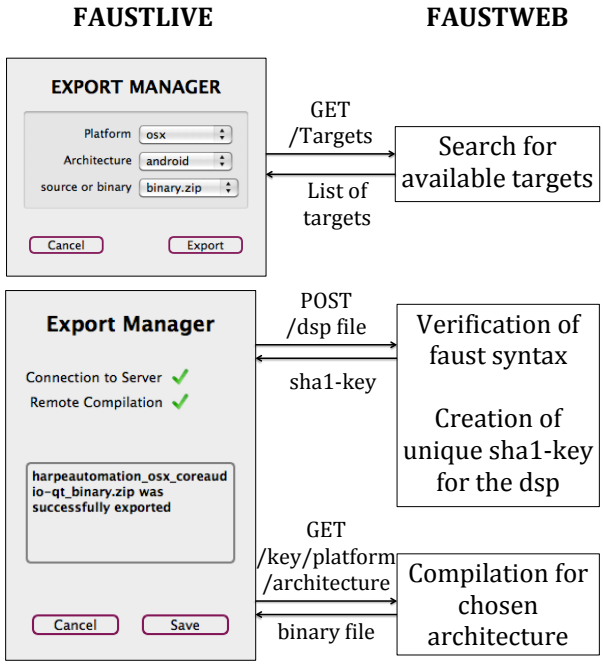

Figure 16: Steps of the compilation chain through FaustWeb

4.6 Session Management

A concept of session is introduced to preserve the state of the application (parameters values, position on screen, audio connections, compilation options, ...) when the application is closed or when the user takes a snapshot, which saves his session in a .tar file.

A FaustLive snapshot is self-contained. All the local resources needed (like FAUST DSPs) are copied into the folder. Pointers to the resources are used as much as possible. But if a source file is erased or the snapshot is transferred on another machine, copies ought to be employed.

To decrease the compilation time, the output of FAUST compiler, the optimized LLVM IR code, is saved. When the application is recalled, FAUST compiler's and LLVM IR to IR optimization steps are skipped. For very heavy programs, the gain can be noticeable (from a few seconds to almost instantaneous).

5 Conclusion

FaustLive brings together the convenience of a standalone interpreted language with the efficiency of a compiled language.

FaustLive offers currently the shortest development cycle for FAUST applications, allowing even to modify the code of an application while it is running. It integrates advanced remote computation and control features for real-time distributed audio applications. Moreover FaustLive provides, via its export functionality, a convenient front-end for FaustWeb, the compilation web service of FAUST. The project is open-source and available on Sourceforge [3]. It runs on Linux, OSX and Windows.

Acknowledgments

This work has been implemented for one part under the INEDIT project [ANR-12-CORD-0009] and for the other part under the FEVER project [ANR-13-BS02-0008]. It is supported by the "Agence Nationale pour la Recherche" .

References

[1] A. Graef. Functional signal processing with pure and faust using the llvm toolkit. 2011.

[2] faust2 repository. http://sourceforge.net/p/faudiostream/code/ci/faust2/tree/.

[3] faustlive repository. http://sourceforge.net/p/faudiostream/faustlive/ci/master/tree.

[4] S. Letz, Y. Orlarey, and D Fober. Comment embarquer le compilateur faust dans vos applications? 2013.

[5] S. Letz, Y. Orlarey, and D Fober. Dynamic compilation of parallel audio applications. 2013.

[6] Y. Orlarey, S. Letz, and D. Fober. Faust: an efficient functional approach to dsp programming. 2009.